Toward Computer Generated Folk Music Using Recurrent Neural Networks

By
Rohan E. Weeden

A Project Submitted in Partial Fulfillment of the Requirements
for the Degree of

Master of Science
in
Computer Science

University of Alaska Fairbanks
May 2019

Approved:

Orion Lawlor, Committee Chair
Glenn Chappell, Committee Member
Jon Genetti, Committee Member
Christopher Hartman, Chair *Department of Computer Science*

Abstract

In this paper, we compare the effectiveness of two different types of Recurrent Neural Networks, fully connected and Long Short Term Memory, for modeling music compositions. We compare both the categorical accuracies of these models as well as the quality of generated compositions, and find that the model based on Long Short Term Memory is more effective in both cases. We find that the fully connected model is not capable of generating non repeating note sequences longer than a few measures, and that the Long Short Term Memory model can do significantly better in some cases.

Contents

1 Introduction

Humans have been composing music for many centuries and the art of music is considered innately human. However, through centuries of study, we have been able to identify characteristics of music that sound good to us, and construct a set of rules that we call "music theory" which can be used as a tool for composing music on paper without needing to experiment, or play it out loud. The existence of music theory indicates that it may be possible to create a general model of music, or of a certain style of music, which could be used to create new, plausible compositions completely algorithmically.

In this project, we are interested in evaluating modeling techniques as applied to Irish folk music that can be used to generate new plausible compositions. Irish folk music has the advantage of being relatively short when compared to classical music, with a lot of repetition. It is also played in many parts of the world, and available in a standard digital form that can be used for training.

2 Prior Work

There has been a considerable amount of research put into music modeling, both using statistical models and neural networks [1]. This section will describe some of the existing modeling approaches as well as different data representations for music.

2.1 Modeling Techniques

Hidden Markov Models (HMMs) are a popular form of temporal modeling for predicting the current state based on a fixed number of previous states. They have seen frequent applications for generating musical accompaniments based on a melody.

One particular area of study where HMMs have been applied is in composition of harmonies based on an existing melody. The MySong application allows users to record themselves singing a melody, and will compose a variety of possible chord harmonizations to that recording [2]. The system analyzes the audio track to determine what pitch distributions occur at each measure, and then chooses chords which appear frequently with a similar pitch distribution in the training data. MySong also allows the user to tweak parameters which affect stylistic elements of the harmonization, such as how happy/sad it should sound, or how traditional/jazzy.

HMMs have also been used to compose melodies. The SuperWillow application composes melodies and accompaniments based on a collection of sample compositions [3]. It uses several HMMs for modeling different aspects of music (chord structure, rhythm, melody). Van Der Merwe and Schulze note however, that HMMs are not particularly well suited to learning long term structure, and as a result, SuperWillow is not capable of composing plausible melodies that

are longer than 8 measures. For short melodies results showed that some of the compositions (38%) were good enough to be mistaken for a human composition.

An alternative to HMMs for statistical modeling are Neural Networks. One of the most promising types of Neural Network for music applications is the Long Short Term Memory (LSTM) neural network [4]. Traditionally, Neural Networks have struggled with the same problem of learning long term structure as HMMs have, largely due to the vanishing gradient problem. LSTM networks however, introduce a way of preserving an internal state with constant error flow, allowing them to better model long term dependencies.

One of the first applications of LSTMs for composing music was in composing bebop-jazz [5]. Eck and Schmidhuber train their LSTM model both on chord progressions alone, and on chord progressions with melody lines. They find that the model is able to both learn both local and global structure of the training dataset, demonstrating that LSTMs are more fit for music composition than traditional Recurrent Neural Networks.

One recent application where we have seen LSTMs used is in composing harmonies based on existing melodies. DeepBach is an LSTM based model for re-harmonizing Bach chorales [6]. DeepBach takes a fixed soprano melody and generates 3 underlying harmonies (alto, tenor and bass) to form a full 4 part chorale. Hadjeres and Pachet find that about 50% of respondents surveyed in a turing test, incorrectly attributed DeepBach compositions to J.S. Bach. They also analytically find that the compositions closely follow classical music conventions, although there are some mistakes.

We have also seen LSTMs applied to melodic composition. Sturm et al. use large LSTM models to compose melodies in the style of traditional Irish tunes [7]. Their char-rnn is trained on character transcriptions of tunes in a format called ABC notation, and is capable of generating full plausible compositions. These compositions follow many of the same patterns as are observed in the training data, but also include unique elements.

LSTMs show an ability to learn musical structures even through an additional layer of encoding. Choi, Fazekas and Sandler train an LSTM model to learn Jazz chord progressions from a text representation [8]. Chords are notated as ascii text (e.g. `D:min7` or `F#:(1,3,b5,b7,9,13)`) and compositions are wrapped inside of start/end markers. Their model is able to produce original chord progressions which often follow well known patterns such as `II-V-I`. This work shows that it is possible for LSTM models to learn musical structure from the notation without needing to directly relate the notation to audible pitches.

2.2 Data Representations

There are many different ways to represent music for training a statistical model. Often the most convenient is a form of text based encoding, because there are large musical datasets available in such encodings. Sturm et al. find success training a model on an ABC notation dataset [7], while Choi, Fazekas and Sandler train their model using .xlab format [8].

Using text representations can add an additional layer of abstraction from the underlying music because every pitch can have multiple notational representations depending on its function in the composition. Eck and Schmidhuber use a vector encoding where each element is either 1 or 0 denoting whether or not the pitch is being played [5]. This allows them to represent polyphonic music by activating multiple vector elements at the same step in time. Hadjeres and Pachet use a midi like encoding where each pitch is represented as a unique integer and the model learns relationships between pitches through an embedding layer [6]. These approaches remove notational elements which make the dataset simpler, but also gets rid of some functional information in the compositions.

3 Our Approach

When deciding how to train our model we have to consider two main factors: the format of our training data, and the architecture of our model. We will discuss data format first as it is arguably the most important factor in successfully training a model. We will discuss features of the raw dataset and how we pre process it before sending it to our model. Finally we will discuss our model's architecture.

3.1 Dataset

We chose to use crowdsourced data from a website called "The Session" (currently located at www.thesession.org). The Session is dedicated to collecting transcriptions of Irish Folk music being played at jam sessions across the globe. It collects these transcriptions by allowing anyone to sign up and post tunes in ABC notation. ABC is a musical notation using mostly ASCII text although some UTF-8 characters such as ♭, ♯, and ♮ are allowed. At the time of this writing, the latest ABC specification can be found here: `https://abcnotation.com/wiki/abc:standard:v2.1`. This data is available for download on GitHub, the raw dataset used in this project is available here: `https://github.com/Askaholic/TheSession-data`

This dataset contains 31,720 transcriptions of tunes. We note that many of these are variations of the same tune, but transcribed by different people. The tunes are categorized by time signature, and key. Choosing a single time signature and key to focus on will allow us to drastically reduce the amount of information that our model needs to learn, meaning shorter training times and higher prediction accuracy.

As shown in table 1, the majority of tunes are either reels or jigs which have the 4/4 and 6/8 time signatures respectively. The majority of traditional tune melodies consist of a sequence of 8th notes, meaning that reels will have eight 8th notes per measure, while jigs will only have 6 per measure. For this project we chose to focus on jigs because there are a significant number of them in the dataset, and they have fewer notes per measure than reels. Fewer notes

Tune Type	Count
barndance	1589
hornpipe	2315
jig	7835
mazurka	421
polka	2310
reel	11774
slide	740
slip jig	1189
strathspey	961
three-two	291
waltz	2295
total	31720

Table 1: Tune distribution by type

means that the patterns we wish for our model to learn will also be shorter, and therefore require a smaller model, and less training time.

In order to make the structure more obvious we remove any tunes according to a number of criteria. We remove tunes with pickup notes, which are extra notes added before the beginning of a phrase as a lead in. Notation wise, these obstruct the regular measure lengths, because they are notated as incomplete measures. We also remove tunes with variant endings, and tunes which are notated using non-ascii characters. This prevents our alphabet from being polluted by characters which do not appear frequently enough in the training data to be learned.

We also pick a single key to train on for complexity reasons. The model will not need to learn the generalization of scale degrees if we limit it to training only on tunes in a single key. Sometimes, this is done by transposing the entire dataset into a single key, however with Irish folk music, the melodies often follow note patterns that are physically easy to play on the fiddle and flute. For example, some tunes which are easy to play on the fiddle in D major, may be a lot harder to play in C major because they would require additional string crossings. Therefore, each key will have its own style originating from the physical process of playing an instrument.

We want to preserve each key's unique style, so we will choose not to transpose tunes, but instead train on only tunes from the most common key. As shown in table 2, the most common key for our dataset of jigs without pickup notes, variant endings, and non-ascii characters is D major.

Our final dataset consists of 433 jigs in the key of D major, all of which have a simple notational structure that is easy to parse and have a clear measure length.

Key	Count
Adorian	99
Amajor	99
Aminor	63
Amixolydian	89
Bdorian	4
Bminor	62
Cdorian	3
Cmajor	30
Ddorian	16
Dmajor	433
Dminor	25
Dmixolydian	86
Edorian	69
Emajor	7
Eminor	124
Emixolydian	6
Fdorian	1
Fmajor	22
Gdorian	10
Gmajor	412
Gminor	12
Gmixolydian	7

Table 2: Tune distribution by key for clean jigs

3.2 Data Pre Processing

Our dataset is initially in ABC format which is primarily designed to convey notation. This means it contains extra information which affects how the music should be rendered, but doesn't change how the music should sound. We are primarily interested in our model learning what makes an Irish folk melody, and not how it is notated, so we transform our ABC tunes into a format that removes much of the notational embellishments.

We start by tokenizing the tune into either `Note` or `Bar` tokens so that we have a sequence of notes with interspersed bar tokens to indicate measures. Notes are identified by a note name (`a-gA-G`), an octave modifier in case the note name does not represent the correct octave, a duration representing the length in eighth notes, and an accidental (`^_=`). For example the sequence of 4 characters `^A,3` would become a single token containing the same information `Note(A,^3)`. Note that this representation still encodes some notational choices, as enharmonic pitches (such as A♯ and B♭) will be considered different tokens. `Bar` tokens on the other hand are all considered to be the same. ABC allows various styles of bar lines for notational purposes, but they all convey the same musical meaning, which is the end of some sequence of notes. We currently

ignore repeat signs because parsing them is not always straightforward, but we keep this in mind as a possible path for continuing this work in the future.

We then expand any `Note` tokens with duration longer than one eighth note into an appropriate number of repeated eighth notes. So for example `Note(A3)` becomes `Note(A), Note(A), Note(A)`. This ensures that for jigs, our sequence will always contain 6 note tokens surrounded by 2 bar tokens. As a result, the musical structure of having 6 eighth notes per measure will be apparent to our model.

In order to feed these tokens to our model, we first convert them to a numerical form. We start by tokenizing our entire dataset and then counting the number of unique tokens that we get. This is referred to as our model's vocabulary and is summarized in table 3 [1].

Token	**Count**	**Token**	**Count**
Bar	8699	Note(C♮)	32
Note(d)	8684	Note(g♯)	29
Note(A)	7827	Note(F♯)	24
Note(B)	5309	Note(f♮)	23
Note(e)	5032	Note(G♯)	19
Note(f)	4989	Note(A♯)	12
Note(F)	4543	Note(c'♮)	8
Note(D)	3473	Note(d')	8
Note(c)	3016	Note(d♯)	7
Note(E)	2612	Note(D')	6
Note(G)	2484	Note(e♯)	5
Note(a)	2134	Note(E♯)	4
Note(g)	2003	Note(C♯)	4
Note(b)	391	Note(F♮)	4
Note(c♮)	171	Note(d♭)	3
Note(A,)	140	Note(G♮)	2
Note(C)	138	Note(G,)	2
Note(B,)	100	Note(D♯)	1
Note(z)	63	Note(d♮)	1
Note(f♯)	45	Note(g♮)	1
Note(c♯)	42	Note(c')	1

Table 3: Model Vocabulary

We then encode each token as a one-hot vector representing it's index in the vocabulary. The most common note becomes `<1, 0, 0, 0 ...>` the second most common becomes `<0, 1, 0, 0 ...>` etc. This new sequence of 42 element vectors becomes the input to our model. Similarly, our model can output a one-hot vector of 42 elements which we then decode back into a note token.

[1] `Note(z)` represents an eighth note rest

3.3 Model

Our model is a recurrent neural network which takes a sequence of one-hot vector encoded tokens (as described above) and predicts the next token as a one-hot vector. The neural network consists of a feature embedding layer, then two computational layers, then a dense output layer. We also intersperse a few dropout layers for regularization. We train two models which use different computational layers, one with densely connected layers, and one with Long Short Term Memory (LSTM) layers. The full Keras model summary for each of the networks is listed in tables 4 and 5. Note that the output shape is a multidimensional array with the batch size (1) as the first dimension, and the number of nodes in the last dimension. For layers with a third dimension, the middle dimension represents the number of previous tokens to use for predicting the next token.

Layer (type)	**Output Shape**	**Param**
embedding (Embedding)	(1, 8, 10)	420
dense (Dense)	(1, 8, 55)	605
dense_1 (Dense)	(1, 8, 55)	3080
dropout (Dropout)	(1, 8, 55)	0
flatten (Flatten)	(1, 440)	0
dense_2 (Dense)	(1, 42)	18522
activation (Activation)	(1, 42)	0
dropout_1 (Dropout)	(1, 42)	0
Total Params		22,627
Trainable params		22,627
Non-trainable params		0

Table 4: Dense Network Architecture

Layer (type)	**Output Shape**	**Param**
embedding (Embedding)	(1, 8, 10)	420
lstm (LSTM)	(1, 8, 30)	4920
lstm_1 (LSTM)	(1, 8, 30)	7320
dropout (Dropout)	(1, 8, 30)	0
flatten (Flatten)	(1, 240)	0
dense (Dense)	(1, 42)	10122
activation (Activation)	(1, 42)	0
dropout_1 (Dropout)	(1, 42)	0
Total Params		22,782
Trainable params		22,782
Non-trainable params		0

Table 5: LSTM Network Architecture

We have chosen the sizes of our computational layers in order to keep the number of trainable parameters for each network as close as possible. We settled on a size of 55 Dense nodes or 30 LSTM nodes by trying a few different sizes and observing that higher amounts of nodes increased computation time, but did not have any noticeable effects on validation accuracy (although a larger LSTM network was able to obtain a better training accuracy).

4 Experimental Design

We wish to determine whether our neural network using LSTM layers will produce a better model for our dataset of Irish jigs than our neural network using only densely connected layers. To do this, we train each of the models on the same dataset for 500 epochs. We then evaluate the two best performing models against the same validation dataset and record the proportion of correctly predicted characters. Using these values we conduct a two proportion z-test to determine if the proportions are significantly different. We also perform a qualitative comparison of the output generated by the two models when recurrently fed their own output.

4.1 Training

In order to evaluate our model's effectiveness we need to split our dataset into training and validation subsets. Our dataset is initially sorted in alphabetical order of tune name. Since tune names are arbitrary we simply take the first 10% of alphabetically sorted tunes as our validation dataset, and the remaining tunes become our training dataset. Splitting up our dataset in this manner has the advantage that two different versions of the same tune will always appear in the same partition, since the two different versions still have the same name. For example, the tune "Bobby Gardiner's" was uploaded by two different people, and therefore appears in our dataset twice. However, both transcriptions get partitioned into the validation dataset because we have sorted tunes in alphabetical order. This guarantees that if our model correctly predicts many notes in "Bobby Gardiner's" we know that it has done so because it has generalized well to folk music, and not because it has seen "Bobby Gardiner's" before in the training data.

With our dataset prepared we are ready to train our models. There are a few quirks about LSTM networks which mean we have to set up our training carefully in order to get meaningful results. LSTM layers are stateful, meaning that they have an internal and mutable state which needs to be preserved across training iterations. This means that the model will learn something about the order in which it sees training samples. Since the order of notes in a tune matters, we need to feed our training samples to our model in the same order that they appear in the dataset. However, once we reach the end of a tune, the tune that follows it is completely arbitrary, so we don't want our model to learn anything about the order in which it sees whole tunes. Therefore, we reset

the model back to its initial state everytime the end of a tune is reached in the training dataset.

4.2 Evaluation

We evaluate our models in two stages. First we evaluate them against the validation dataset at the end of each training epoch. This lets us track how the model is improving with each pass through the training data. Then we take our highest performing models and qualitatively compare the new compositions that they create when fed their own output recurrently.

During the first stage of evaluation, each sample sequence of notes from the training data is passed to the model, and we record if the model predicts the next note correctly. The predicted output of an evaluation sequence is not used as input for the following sequence. This means we are effectively evaluating how good the model is at predicting what notes should appear in existing music.

Once we have collected the proportions of correctly predicted notes for each of our best performing models, we compare them using a z-statistic. We are interested in showing that the proportions are significantly different ($p_1 \neq p_2$), so we conduct a two-tailed test using the following equations.

$$\text{Calculate a pooled sample proportion: } p = \frac{p_1 \cdot n_1 + p_2 \cdot n_2}{n_1 + n_2}$$

$$\text{Calculate the standard error: } SE = \sqrt{p(1-p) \cdot (\frac{1}{n_1} + \frac{1}{n_2})}$$

$$\text{Calculate the z-score: } z = \frac{p1 - p2}{SE}$$

Our validation data consists of 4,560 samples of 8 note sequences and since both models were evaluated on the same validation data, our n1 and n2 are both 4,560. For the outcome of this test see the Results section below.

During the second stage of evaluation, we use our model to generate a new composition from an existing seed. The seed is chosen as the first 8 tokens of a tune from our validation dataset. Then we predict the next character using our model and append that to our final composition. We use the method of recurrently feeding the last 8 tokens in our composition as input to our model to produce a new token. As our model has no way of signaling that the end of a tune has been reached, we repeat this process for 250 iterations, which is longer than any tune in the training dataset.

Once we have generated a few original compositions from each model, we qualitatively evaluate them by verifying that they have the correct number of notes per measure, and by looking for repeating patterns. Finally we listen to the tunes through an ABC to MIDI converter and determine whether they sound like plausible music.

5 Results

We will first discuss the statistical significance of our models validation accuracy and then do a qualitative comparison between the original compositions created by each model.

5.1 Validation Accurracies

Our densely connected model reached its maximum validation accuracy after 51 epochs and then continued to overfit to the training data. Figure 1 shows the history of both the validation accuracy and training accuracy of our densely connected network for each epoch. At epoch 51 the model achieved a 40.26% validation accuracy on 4,560 samples.

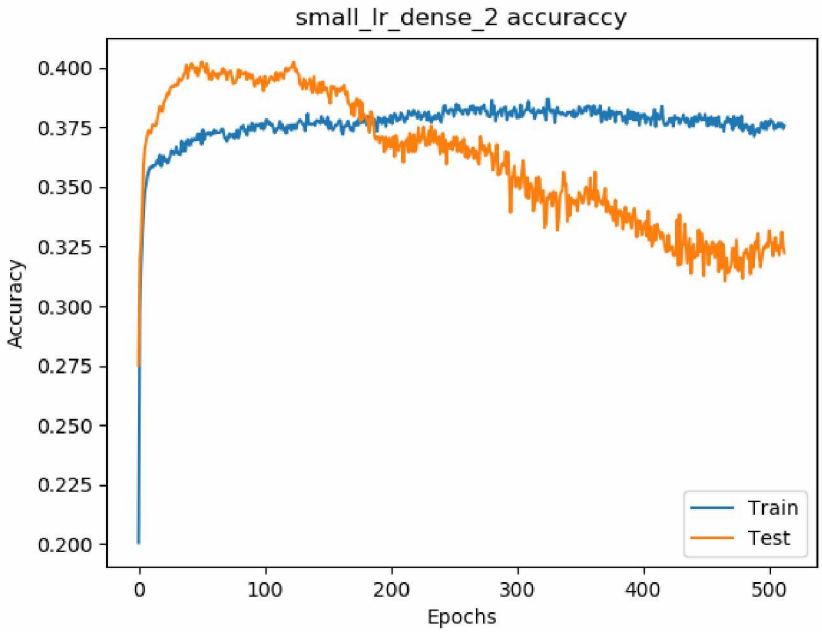

Figure 1: Densely connected model accuracy

Our LSTM took longer to train achieving a maximum validation accuracy after 196 epochs. Figure 2 shows the history of validation and training accuracy for our LSTM model at each epoch. Note that the LSTM does not appear to overfit as much as the densely connected model, and the validation accuracy stays relatively stable over all 500 epochs. At epoch 196 the maximum validation accuracy achieved was 43.46% over 4,560 samples.

We now proceed to determine whether the difference between 40.26% accuracy for the densely connected model and 43.46% accuracy for the lstm model is significant at 95% confidence. First our pooled sample proportion is $p = \frac{(0.4026)(4560)+(0.4346)(4560)}{4560+4560} = 0.4186$. We can use this to calculate our standard error $SE = \sqrt{0.4186(1-0.4186)(\frac{1}{4560}+\frac{1}{4560})} = 0.0103$. Finally our z-score is $z = \frac{0.4026-0.4346}{0.0103} = -3.097$. Since the p-value for a z-score of -3.097 is less than 0.05 we can conclude that there is a significant difference between

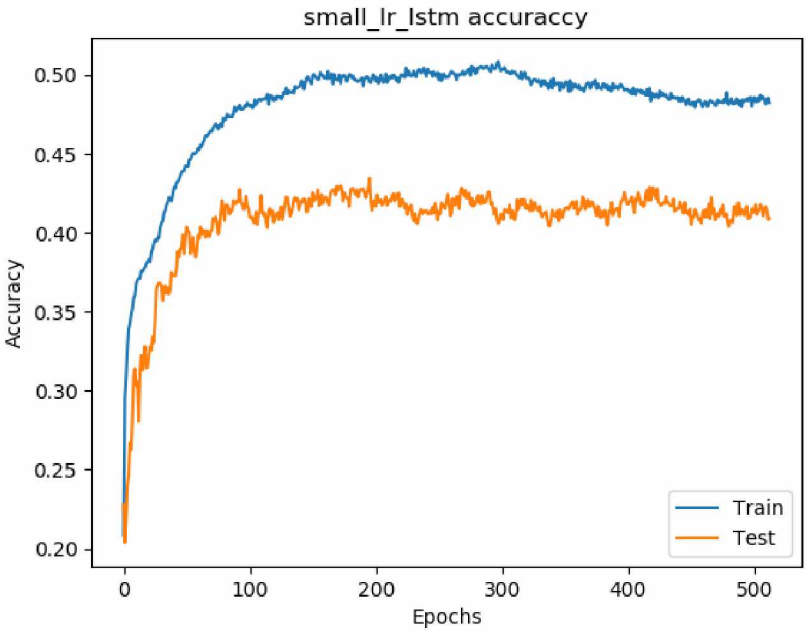

Figure 2: LSTM model accuracy

our models' validation accuracies.

5.2 Original Compositions

Along with determining which of our models does a better job mathematically of modeling folk music, we are also interested in determining if that also translates to creating more plausible original compositions. We note that one approach to evaluating the plausibility of compositions is to conduct a turing test, where a randomly selected group of individuals is presented with a real tune and a computer generated one and asked to guess which is which. This turns out not to be necessary because the difference in quality of compositions between the two models is glaringly obvious.

To demonstrate, we first present sample of compositions from the densely connected network. Note that the first 8 tokens in the sample are a seed from a real tune in our validation dataset. This seed was always chosen to be the first 8 tokens in the tune. The use of ... indicates that the previous 23 tokens continued to repeat and that we truncated them for brevity.

1. `|AAddcd|AAdddd|fffedd|fffedd|fffedd|...`
2. `A,|DDDFEF|DDDDDD|DDDDDD|DDDDDD|...`

3. A,A,DDEF|EEEEEE|DDDDDD|DDDDDD|DDDDDD|...

4. ffedcB|ABAAFA|AFAAFA|AFAAFA|AFAAFA|...

5. Adddce|dddddd|dddddd|dddddd|...

None of the compositions from the densely connected model were able to produce non-repeating sequences that were longer than only a few measures. However, the compositions always had the expected jig structure of bar lines separating groups of 6 notes.

The LSTM model on the other hand, was able to produce much longer sequences of non-repeating patterns. Again note that the first 8 tokens are a seed from the validation dataset. The first two compositions are included until the last complete measure since 250 iterations ended in the middle of a measure.

1. |ddcBBA|Bcdefg|fedeed|edBBAF|DEFABA|BAFAFE|DEFABB|AFEDDD
   |dddddd|dcBAFE|dddefe|dddBBB|defgfe|dddddе|feddef|edBBAF
   |DEFABA|BAFAFE|DEFABB|AFEDDD|dddddd|dcBAFE|dddefe|dddBBB
   |defgfe|dddddе|feddef|edBBAF|DEFABA|BAFAFE|DEFABB|AFEDDD
   |dddddd|dcBAFE|dddefe|dddBBB|

2. |AAddcd|BBBBBB|gecddd|edcddd|fefgec|dedddd|dddddd|dcBAFE
   |dddefe|dddBBB|defgfe|dddddе|feddef|edBBAF|DEFABA|BAFAFE
   |DEFABB|AFEDDD|dddddd|dcBAFE|dddefe|dddBBB|defgfe|dddddе
   |feddef|edBBAF|DEFABA|BAFAFE|DEFABB|AFEDDD|dddddd|dcBAFE
   |dddefe|dddBBB|defgfe|dddddе|

3. A,A,DDEF|EEEEFA|BBBBBB|ABcddA|BBBBBB|defaaa|aaaffe|fBBBAB
   |BBBBBB|BG^GGBB|BBBBBB|BBBcBd_dABcddddddddddddddddddddddddd...

4. ffedcB|ABAFFE|DDDDDD|DDDDDD|DDDDDD|...

5. |DFAddd|cBcABA|dfedBB|AABcBB|BBBBBB|deffed|eeeeee|fedBBA
   |Bddddd|BBBBBB|gedBBA|BGEEFE|DDDDDD|DDDDDD|DDDDDD|...

6. ddDFED|=c'=c'CdAF|DEFAAF|FEDDDD|DDDDDD|DDDDDD|DDDDDD|...

Looking at the first two compositions, we notice that they both eventually fall into the same repeating 12 measure pattern which we have extracted here:

```
|dddddd|dcBAFE|dddefe|dddBBB
|defgfe|dddddе|feddef|edBBAF
|DEFABA|BAFAFE|DEFABB|AFEDDD
```

Both of these tunes started with different seeds which generated an original melody line for a few measures before entering this 12 measure sequence. By itself, this 12 measure sequence makes quite a bit of musical sense, although we note that it is either 4 measures shorter or longer than we would expect a musical phrase to be for a jig. We can analyze this phrase and we will notice that it starts and ends on a long d note which makes sense in the context of D

major. We also notice that the melody starts in the higher octave, and resolves to the lower octave which makes the sequence feel like it is progressing.

Even though the LSTM generated much more plausible tunes with the first two seeds, for seeds 4 and 6, it only did slightly better than the dense model tended to do, managing 1-3 unique measure before getting stuck in a loop of repeating `D` notes. Composition 6 is also completely implausible as a traditional tune primarily because of the 2 octave jump from the high `c'` to the low `C` in the second measure. Not only is a jump like this difficult to play on most instruments, but it also produces an unpleasant sound considering that C natural does not naturally occur in the key of D major.

For seed 3 we also notice that the LSTM completely stopped generating bar lines after 10 measures. Although this composition is a great example of the internal model state being used to predict different notes for the same input sequence `|BBBBBB|`. We also see the model predicting an `_d` or d♭ token which only has 3 occurrences in the training data when it should have predicted a bar line. We speculate that this is probably due to the one hot encoding having to choose between several similarly activated outputs, and the one for d♭ being strongest due to a lack of appearances in the training data. After the input sequences lose the bar lines we notice that the network is unable to recover.

6 Conclusion and Future Work

It is clear that our LSTM model out performed the densely connected model over all. The LSTM was able to achieve a better prediction accuracy on existing tunes, and it was also able to generate long sequences of non-repeating notes for some input seeds. We also saw evidence that the LSTM model was using its internal state when making note predictions.

We would like to expand this work to be able to learn additional keys and time signatures. One interesting idea is to see if we can create a model which can compose tunes in a variety of keys without mixing keys in any individual composition. We note that in order to do experiments like this, we will need to adjust our data representation in order to encode additional information.

We would like to use a simple ABC text representation instead of parsing the music into a more regular format because ABC notation has the advantage that it can encode almost any musical information at all. This means a model capable of learning music from ABC notation could potentially be used to learn arbitrary music styles, time signatures, keys, etc. given enough training data. At first we were using such a representation, but were unable to achieve meaningful results, however, we believe that this was due to other factors which would have caused any of our attempts to fail regardless of the data representation. In our case, we were originally using a default learning rate which was too high for our model, causing our loss function to increase instead of decrease with each epoch.

There may be some opportunity to combine LSTM Neural Networks with a Generative Adversarial Network (GAN) approach [9]. GANs attempt to use the

progress that has been made in discriminative Neural Networks to help drive a generative Neural Network, potentially improving the generative network's ability to create data which closely resembles the training data, but is not an exact copy.

References

[1] N. Collins, "The analysis of generative music programs," *Organised sound*, vol. 13, no. 3, pp. 237–248, 2008.

[2] I. Simon, D. Morris, and S. Basu, "Mysong: automatic accompaniment generation for vocal melodies," in *Proceedings of the SIGCHI conference on human factors in computing systems*, pp. 725–734, ACM, 2008.

[3] A. Van Der Merwe and W. Schulze, "Music generation with markov models," *IEEE MultiMedia*, vol. 18, no. 3, pp. 78–85, 2011.

[4] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[5] D. Eck and J. Schmidhuber, "A first look at music composition using lstm recurrent neural networks," *Istituto Dalle Molle Di Studi Sull Intelligenza Artificiale*, vol. 103, 2002.

[6] G. Hadjeres, F. Pachet, and F. Nielsen, "Deepbach: a steerable model for bach chorales generation," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 1362–1371, JMLR. org, 2017.

[7] B. L. Sturm, J. F. Santos, O. Ben-Tal, and I. Korshunova, "Music transcription modelling and composition using deep learning," *arXiv preprint arXiv:1604.08723*, 2016.

[8] K. Choi, G. Fazekas, and M. Sandler, "Text-based lstm networks for automatic music composition," *arXiv preprint arXiv:1604.05358*, 2016.

[9] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, pp. 2672–2680, 2014.

7 Appendix

lstm.py

```
import argparse
import json
import random
import sys
```

```
import tensorflow as tf
from tensorflow import keras
import numpy as np
from data_prep import jigs, replace_mapping
from generators import KerasTuneGenerator
import os
import pickle

# True if the model predicts window size characters,
    False if it only predicts one
PREDICT_SEQUENCE = False
NUM_FEATURES = 10
USE_LSTM = True


def read_chars(filename):
    with tf.gfile.GFile(filename) as f:
        return list(f.read())


def create_model(vocab_size, input_length, dropout=True):
    if USE_LSTM:
        model = keras.Sequential([
            keras.layers.Embedding(vocab_size,
                NUM_FEATURES, input_length=input_length,
                batch_input_shape=(1, input_length)),
            keras.layers.LSTM(30, return_sequences=True,
                stateful=True),
            keras.layers.LSTM(30, return_sequences=True,
                stateful=True),
            keras.layers.Dropout(0.1),
            keras.layers.Flatten(),
            keras.layers.Dense(vocab_size),
            keras.layers.Activation('softmax'),
            keras.layers.Dropout(0.1),
        ])
        lr=0.0001
    else:
        model = keras.Sequential([
            keras.layers.Embedding(vocab_size,
                NUM_FEATURES, input_length=input_length,
                batch_input_shape=(1, input_length)),
            keras.layers.Dense(55),
            keras.layers.Dense(55),
            keras.layers.Dropout(0.1),
```

```
            keras.layers.Flatten(),
            keras.layers.Dense(vocab_size),
            keras.layers.Activation('softmax'),
            keras.layers.Dropout(0.1),
        ])
        lr=0.00001
    model.compile(
        loss='categorical_crossentropy',
        optimizer=keras.optimizers.RMSprop(lr=lr),
        metrics=['categorical_accuracy']
    )

    return model


def test_model(model, vocab_size, test_data, decode_dict,
    window_size):
    # print(model.layers[1].get_weights())
    # print(model.layers[2].get_weights())

    # Set this to true for the model to be evaluated on
        an existing tune rather
    # than using it's own output as input
    seed_whole_tune = False

    num_predict = 250
    dummy_iters = random.randrange(0, 20)
    example_training_generator_k = KerasTuneGenerator(
        test_data, window_size, 1, vocab_size)
    example_training_generator =
        example_training_generator_k.generate(
        return_sequence=PREDICT_SEQUENCE)

    print("Testing...", dummy_iters)
    for i in range(dummy_iters):
        next(example_training_generator)
        while not example_training_generator_k.tune_ended
            :
            d = next(example_training_generator)
    pred_print_out = "Predicted chars: \n"
    data_pairs = next(example_training_generator)
    data = data_pairs[0]
    print("Seed: '{}'".format("".join([str(decode_dict[d
        ]) for d in data[0]])))
    for i in range(num_predict):
        if seed_whole_tune:
```

```
            prediction = model.predict(next(
                example_training_generator)[0])
        else:
            prediction = model.predict(data)
        if PREDICT_SEQUENCE:
            predict_word = np.argmax(prediction[:,
                window_size-1, :])
        else:
            predict_word = np.argmax(prediction)
        pred_print_out += str(decode_dict[predict_word])
        data = np.array([np.concatenate((data[0][1:], [
            predict_word]))])
    print(pred_print_out)


def one_hot_to_num(arr):
    for i in range(len(arr)):
        if arr[i] == 1:
            return i


class ResetStateAfterTune(keras.callbacks.Callback):
    def __init__(self, gen):
        super().__init__()
        self.gen = gen

    def on_batch_end(self, batch, logs={}):
        if self.gen.tune_ended:
            self.model.reset_states()


def count_samples(iter):
    samples = 0
    for _ in iter:
        samples += 1
    return samples


def train_model(manager, model, train, valid, vocab_size,
    window_size, epochs, progress_name):
    batch_size = 1

    train_gen = KerasTuneGenerator(train, window_size,
        batch_size, vocab_size)
    valid_gen = KerasTuneGenerator(valid, window_size,
        batch_size, vocab_size)
```

```
print("Counting samples... ", end="")
sys.stdout.flush()
samples_train = count_samples(train_gen.generate())
print("Training:", samples_train, end='')
sys.stdout.flush()
samples_valid = count_samples(valid_gen.generate())
print(" Validation:", samples_valid)

reset_state_after_tune = ResetStateAfterTune(
    train_gen)
model_history = {
    "loss": [],
    "categorical_accuracy": [],
    "val_loss": [],
    "val_categorical_accuracy": []
}
# The best validation accuracy we've seen so far.
    Whenever we create a model
# that does better than this, save that model to a
    file.
best_val_acc = 0
save_at = 1
for i in range(epochs):
    epoch = i+1
    print("Epoch {}/{}".format(epoch, epochs))
    history = model.fit_generator(
        train_gen.generate(return_sequence=
            PREDICT_SEQUENCE),
        samples_train // batch_size,
        epochs=1,
        validation_data=valid_gen.generate(
            return_sequence=PREDICT_SEQUENCE),
        validation_steps=samples_valid // batch_size,
        shuffle=False,
        callbacks=[reset_state_after_tune] # Only
            required for LSTM
    )
    val_acc = history.history['
        val_categorical_accuracy'][0]
    for key in model_history.keys():
        model_history[key].append(float(history.
            history[key][0]))
    model.reset_states()
    if progress_name is not None:
```

```
            with open(os.path.join("training", f"{
                progress_name}-history.json"), 'w') as f:
                json.dump(model_history, f)
            if val_acc > best_val_acc or epoch == save_at
                :
                manager.save_model(f"{progress_name}-{
                    epoch}")
                if val_acc > best_val_acc:
                    best_val_acc = val_acc
                if epoch == save_at:
                    save_at *= 2
    return model_history


class MLManager(object):
    def __init__(self, dataset):
        (
            self.train_gen,
            self.valid_gen,
            self.test_gen,
            self.decode_dict
        ) = jigs.load_data("data", dataset, verbose=0)
        self.vocab_size = len(self.decode_dict)
        self.window_size = 10
        self.model = None
        self.decode_dict[len(self.decode_dict) - 1] = '-'
        self.history = None

    def set_window_size(self, steps):
        self.window_size = steps
        return self

    def create_model(self):
        self.model = create_model(self.vocab_size, self.
            window_size)

    def load_weights(self, checkpoint):
        self.model.load_weights(checkpoint)
        return self

    def test_model(self):
        assert self.model is not None
        test_model(
            self.model,
            self.vocab_size,
            self.test_gen,
```

```
            self.decode_dict,
            self.window_size
        )

    def train_model(self, epochs=10, progress_name=None):
        assert self.model is not None
        self.history = train_model(
            self,
            self.model,
            self.train_gen,
            self.valid_gen,
            self.vocab_size,
            self.window_size,
            epochs,
            progress_name
        )

    def save_model_png(self, name):
        keras.utils.plot_model(self.model, to_file=f'
            training/{name}.png')

    def save_model(self, name):
        model_data = {
            "weights": [l.get_weights() for l in self.
                model.layers
                        if not isinstance(l, keras.layers
                            .Dropout)],
            "mapping": self.decode_dict,
            "history": self.history,
            "config": self.model.get_config()
        }
        with open(os.path.join("training", f"{name}.pkl")
            , 'wb') as f:
            pickle.dump(model_data, f)

    def load_model(self, name):
        with open(os.path.join("training", f"{name}.pkl")
            , 'rb') as f:
            model_data = pickle.load(f)
        # if "config" in model_data:
        #     self.model = keras.Sequential.from_config(
            model_data['config'])
        for i, layer in enumerate(filter(lambda l: not
            isinstance(l, keras.layers.Dropout), self.
            model.layers)):
            layer.set_weights(model_data['weights'][i])
```

```
        self.decode_dict = model_data['mapping']
        self.history = model_data['history']

    def load_model_legacy(self, name):
        nzfile = np.load(os.path.join("training", "{}.npz
            ".format(name)))
        for i, layer in enumerate(self.model.layers):
            if isinstance(layer, keras.layers.Dropout):
                continue
            layer.set_weights(nzfile["arr_{}".format(i)])
        with open(os.path.join("training", "{}-mapping.
            json".format(name)), 'r') as f:
            self.decode_dict = {int(k): v for (k, v) in
                json.load(f).items()}


def print_help():
    print(f"{__file__} dataset [test]")


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('dataset')
    parser.add_argument('-m', '--model', help='The name
        of the model to load/save')
    parser.add_argument('-e', '--epochs', help='The
        number of epochs to train for', type=int)
    parser.add_argument('-p', '--progress', help='Save
        model progress during training',
                        action='store_const', const=True,
                            default=False)
    parser.add_argument('mode', choices=['train', 'test'
        ])

    args = parser.parse_args()

    dataset = args.dataset
    model_name = args.model or dataset
    epochs = args.epochs or 5
    save_progress = args.progress

    manager = MLManager(dataset) \
        .set_window_size(8)

    manager.create_model()
```

```
    if args.mode == "test":
        manager.load_model(model_name)
        manager.model.summary()

        manager.test_model()
    else:
        manager.model.summary()
        progress_name = None
        if save_progress:
            progress_name = model_name
        try:
            manager.save_model_png(model_name)
        except ImportError:
            print("WARNING: Graphviz not installed,
                skipping image generation...")
        manager.train_model(epochs=epochs, progress_name=
            progress_name)
        manager.save_model(model_name)
        manager.test_model()


if __name__ == "__main__":
    main()
```